Assignment Two: Build a game AI

Sunday, April 17, 2016

Games And AI

Samuel McKay

2016

Abstract

One of the more complex elements in a modern game is creating believable artificial intelligence (AI) for non-playable characters (NPCs).

[Introduction 3](#_Toc448841823)

[Game and AI type 4](#_Toc448841824)

[Background Theory 5](#_Toc448841825)

[Finite State Machines 5](#_Toc448841826)

[Methodology 7](#_Toc448841827)

[3rd Party additions 8](#_Toc448841828)

[Conclusion 9](#_Toc448841829)

[References 10](#_Toc448841830)

[Appendix 11](#_Toc448841831)

# Introduction

The aim of this project is to create a fully-fledged game with AI.

# Game and AI type

The game created for this project is a first person shooter with a “hostage rescue” mission. It involves two types of a AI, a complex enemy AI agent and a simpler hostage agent. The enemy agents’ have 6 different behaviours compared to the hostage’s 2 behaviours. To add some complexity to the game there are two ways of completing the mission, the player can either use a combination of stealth and observation to free the hostage silently or they can for an all-out assault and attempt to shoot their way to freeing the hostage. This allows for the addition of stealth mechanics such as a crouching and a “hearing” mechanic for the enemies.

# Background Theory

## Finite State Machines

A large part of modern AI is the use of a Finite state machine (FSM). Finite state machines are simply a series of states connected together by transitions. A good example of this is an elevator where each floor is a different state and a button press causes the transition to a different state. In terms of game development FSMs are often used to represent the status, or the “brain”, of a non-playable character (NPC). Events within the game can be used to trigger a state transition, these states can vary from spotting a player to noticing a dead guard or even simply the time of day. Similar to the elevator example, not all states are linked to all other states. The lift cannot go from floor 1 -3 without passing floor 2. FSMs are usually represented with diagrams such as figure 1, which allows a developer to easily see which state should link together and how they transition between the states.

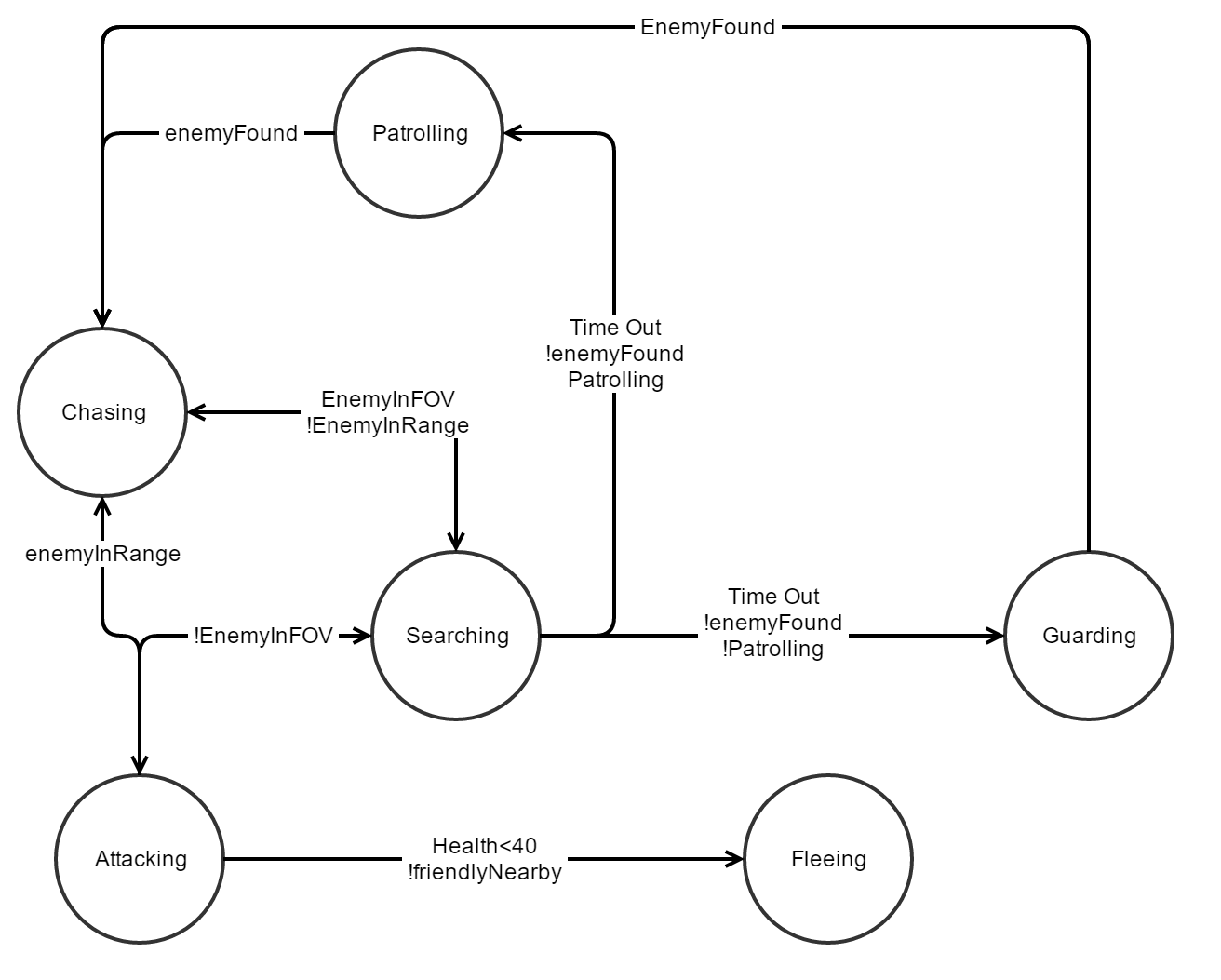


Figure Finite State Machine for this project

State machines in large games can easily get very complicated so layers of abstraction are often added. For example, having a state machine within a state machine, where at the top level its state may be “Attack” but within that is another state machine to establish how to attack the enemy. This additional state machine could allow it to take cover or to move closer to the enemy before finally firing shots at them.

## State Machine Design

Designing a complex state machine can be a long task but is one that is necessary early on, since drastically changing the state machine further down the line could have severe knock effects on the rest of the game. Such as causing bugs in the interactions between the AI and player or world items. As aforementioned the game created for this project is a first person shooter (FPS) which allows the player a choice in how to complete the mission, stealthy or loud. In order to create an artificial intelligence that automatically and dynamically adapts to both of these playstyles without the player making a direct choice requires a fairly complex and broad state machine (Figure 1).

As such the enemy state machine comprises of 6 individual states, despite 6 states existing in the FSM most AI agents will only access 5 of them. This is due to the enemy guards either guarding or patrolling, a feature that again is fairly dynamic and will be explained further in the methodology section. This allows the enemy to appear to have 2 behaviours from 1 state machine without repeated code, good for readability, maintenance and memory usage.

In addition to this an additional much less complex state machine was created to control the hostage character. This simply consists of two states, released and captured, to allow the player to free the hostage and introduces the challenge of keeping the AI from stopping the hostage escape.

# Methodology

## Finite State Machine

The most complex state machine in the game consists of 6 different states, as such a fairly robust method was required in order to reduce the amount of repeated code. Initially it was decided that using an Interface (REF) would be the best way to do this since all states implementing that interface needed to use the same methods. Each state would then implement this interface, allowing the AI controller class to call the below function. This meant changing state was a case of assigning the new state to current state.

void FixedUpdate()

{

currentState.updateState();

}

However, interfaces were later found wasteful as many states shared functions but there was no once place that could hold all the functions and be accessible by all states. Refactoring the code to use a base class to implement the shared functions and members meant a drastic reduction in the amount of repeated code, massively limiting the scope for bugs and easing the maintenance of the code.

Another advantage of this method meant that if a variation of a particular function was required, the specific state requiring it could override the base method in order to create its own version. This method was beneficial since functions like “updateState” could be localised to each state. The controller class then didn’t need to know which state was being updated, an Object Orientated Programming method known as polymorphism.

## Enemy

The enemy state machine contains 6 states although, as previously explained, only 5 are currently used by an enemy guard. I.e. a guard with a patrol won’t use the guard state and vice versa. However, it is possible for a guard to be given a patrol at runtime, in which case they would automatically start using the patrol state instead.

The enemy units currently roam about their assigned patrols or stay in their allocated guard position until they either see or hear the player. At which point they will then begin chasing the player until they are within range and can open fire.

MORE?

### Patrol Point system

Alongside the enemy’s FSM a patrol point system has been added to the game. This allows developers to drag and drop waypoints on to the map (ensuring their positions are whole numbers). The developer can then drop multiple waypoints on to the enemy guard’s *Enemy\_Controller* Patrol points variable in the inspector (figure 2). This will automatically generate a patrol for that agent, points can be used by multiple agents and an agent without any points will become a static guard and won’t patrol. Below is how the FSM differentiates between a patrolling guard and a static guard.

public virtual void move()

{

//Debug.Log("ROUTE POS X" + routePos.Value.x);

Vector3 targetPos = new Vector3(routePos.Value.x, enemy.transform.position.y, routePos.Value.y);

Vector3 velocity = Vector3.zero;

enemy.transform.position = Vector3.Lerp(previousPos, targetPos, distance);

enemy.transform.LookAt(targetPos);

if (distance >= 1)

{

if (routePos == route.Last)

{

nextPatrolPoint++;

if (nextPatrolPoint > enemy.patrolPoints.Length)

{

nextPatrolPoint = 0;

}

if (enemy.patrolPoints.Length > 0)

{

patrol();

}

else

{

enemy.guardState.moveToGuardPoint();

enemy.currentState = enemy.guardState;

}

}

else

{

routePos = routePos.Next;

distance = 0;

previousPos = targetPos;

}

}

}

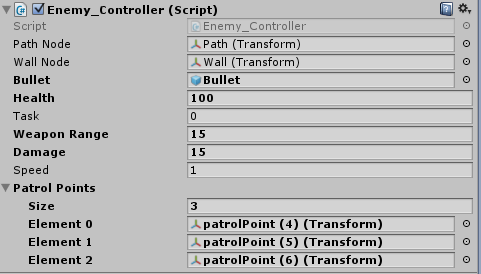


Figure Creating a Patrol

## Hostage

To give the player an objective a new type of AI was created, Hostage. The hostage uses a finite state machine in the same way as the enemy agents except it has different states, captured and released. The FSM is far simpler than then enemy AI, and is less resource intensive since all transitions are trigger based and simply wait for either the enemy or the player to walk close to it. The player can “free” the hostage by getting close to them and ensuring the enemy don’t get so close that they cause the hostage to become captured again. Once the hostage is at the player’s starting point the game is won.

## Group Control

To add further complexity to the game, an enemy group controller was created. This controller was essentially a listener that waited for an enemy under its control to come under attack before then diverting the nearest enemy unit to assist in the firefight. Unfortunately, due to an unknown bug causing the work of the group controller to be overwritten with blank data this has not been implemented into the final version. The source code however is included regardless.

GameObject nearestEnemy = findNearestEnemy(enemy);

if(nearestEnemy == null)//no remaining enemies

{

return;

}

nearestEnemy.GetComponent<Enemy\_Controller>().currentState.newRoute(new Location(Mathf.FloorToInt(enemy.transform.position.x), Mathf.FloorToInt(enemy.transform.position.z)));

nearestEnemy.GetComponent<Enemy\_Controller>().currentState = nearestEnemy.GetComponent<Enemy\_Controller>().patrolState;

## Possible Improvements

Currently the game runs at an average of 250+ frames in debug mode. As such there’s no pressing need to optimize further, however, if optimizations were required, they could be achieved through the use of a scheduler to allow more complex operations to run over several frames. This could include pathfinding. Multithreading could also be used to an extent in order to spread the cost of running several AI agents across multiple cores. This of course would come at the cost of the usual overheads associated with multi-threading.

# 3rd Party additions

Unity3D was used for creating this game, all other assets used are included in Unity’s “standard assets” pack. No other scripts were used.

# Conclusion

In conclusion, the task of creating a fully functional game with artificial intelligence was successful. Not only was a small game created but the framework for many more levels is in place as well as some very useful systems including the patrol points system and the finite state machines. All this helps to create an AI with realistic behaviour and reactions to the player’s actions. The modular design of the code allows it to easily be expanded upon in the future. One disadvantage is the lack of a proper group controller to execute some more complex behaviour such as flanking. The framework for such an entity has been implemented so this feature could be added in the future.



Overall this has been a successful exercise and a lot was learnt about AI techniques and also common elements of first person shooters. It would be very easy to take this project forward and turn it into a releasable game.

# References

http://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

# Appendix